Srinivasan et al.

ProQuest®

Help

Basic Search | Advanced Search | Topic Guide | Publication Search | Marked List : 0 articles

Interface language: English

Databases selected: Multiple databases...

Article View « Back to Results < Previous Article 4 of 6 Next > Publisher Information

Print Email Mark Article Abstract , Full Text , Text+Graphics , Page Image - PDF

Object persistence in object-oriented applications

Srinivasan, V, Chang, D T. IBM Systems Journal. Armonk: 1997. Vol. 36, Iss. 1; pg. 66, 22 pgs

» Jump to full text

Subjects:	Object oriented programming, Studies, Models, Relational data bases, User interface
Classification Codes	5240 Software & systems, 9130 Experimental/theoretical treatment
Author(s):	Srinivasan, V, Chang, D T
Publication title:	IBM Systems Journal. Armonk: 1997. Vol. 36, Iss. 1; pg. 66, 22 pgs
Source Type:	Periodical
ISSN/ISBN:	00188670
ProQuest document ID:	11136835
Text Word Count	12590
Article URL:	http://gateway.proquest.com/openurl?ctx_ver=z39.88-2003&res_id=xri:pqd&rft_val_fmt=ori:fmt:kev:mtx:journal&genre=article&rft_id=xri:pqd:did=00000001

More Like This »Show Options for finding similar articles

Abstract (Article Summary)

There are 3 classes of solutions for implementing persistence in object-oriented applications: 1. the gateway-based approach, which involves adding **object**-oriented **programming** access to persistent data stored using traditional non-object-oriented data stores, 2. the object-relational database-management system (DBMS) approach, which involves enhancing the extremely popular relational data model by adding object-oriented modeling features, and 3. the object-oriented DBMS approach (also called the persistent programming language approach), which involves adding persistence support to **objects** in an **object**-oriented **programming** language. The major characteristics and requirements of object-oriented applications and how they may affect the choice of a system and method for making objects persistent in that application are described.

Full Text (12590 words)

[Headnote]

Object-oriented models have rapidly become the model of choice for programming most new computer applications. Since most application programs need to deal with persistent data, adding persistence to objects is essential to making object-oriented applications useful in practice. There are three classes of solutions for implementing persistence in object-oriented applications: the gateway-based object persistence approach, which involves adding **object**-oriented **programming** access to persistent data stored using traditional nonobject-oriented data stores, the object-relational database management system (DBMS) approach, which involves enhancing the extremely popular relational data model by adding object-oriented modeling features, and the object-oriented DBMS approach (also called the persistent programming language approach), which involves adding persistence support to **objects** in an **object**-oriented **programming** language. In this paper, we describe the major characteristics and requirements of object-oriented applications and how they may affect the choice of a system and method for making objects persistent in that application. We discuss the user and programming interfaces provided by various products and tools for object-oriented applications that create and manipulate persistent objects. In addition, we describe the pros and cons of choosing a particular mechanism for making objects persistent, including implementation requirements and limitations imposed by each of the three approaches to object

persistence previously mentioned. Given that several object-oriented applications might need to share the same data, we describe how such applications can interoperate with each other. Finally, we describe the problems and solutions of how objectoriented applications can coexist with nonobject-oriented (legacy) applications that access the same data.

Object-oriented modeling, design, and programming1-5 have rapidly become the model of choice for programming new computer applications. Since most application programs need to deal with persistent data, adding persistence to objects is essential to making object-oriented applications useful in practice. Before the advent of object-oriented application development, applications typically used relational database management systems (DBMSs) to store their persistent data (and most still do).

Relational DBMSs typically provide support for storing data used in traditional business applications such as banking transactions and inventory control. The relational model6 is the basis of many commercial relational DBMS products (e.g., DB2*, Informix**, Oracle* *, Sybase* *) and the structured query language (SQL)7 is now a widely accepted standard for both retrieving and updating data. The basic relational model is simple and mainly views data as tables of rows and columns. The types of data that can be stored in a table are basic types such as integer, string, and decimal, and other special types such as BLOB (binary large object) and CLOB (character large object). These systems typically do not allow users to extend the type system by adding new data types. They also only support first-normal-form relations7 in which the type of every column must be atomic, i.e., no sets, lists, or tables are allowed inside a column. Relational DBMSs have been extremely successful in the marketplace, growing into an approximately four-billion-dollar market in a decade. These systems are extremely good for a class of applications with simple data models and extensive querying needs. The use of a standard declarative query language in SQL makes it possible for applications to transparently access relational DBMS data from different vendors.

As opposed to the simple data model used by traditional business applications using a relational DBMS, object-oriented applications make extensive use of many new object-oriented features such as a user-extensible type system, encapsulation, inheritance, dynamic binding of methods, complex and composite objects (not first-normal-form objects), and object identity. The limitations of the data models supported by the relational DBMS therefore needed to be relaxed in order to enable the building of more complex (object-oriented) business and nonbusiness applications. As a result, there has been much activity in designing and implementing systems to handle object persistence. Recently it has become clear that there are essentially three major approaches to object persistence: the gateway-based object persistence approach, the object-relational DBMS approach, and the object-oriented DBMS approach (also called the persistent programming language approach). Each of these three approaches to object persistence evolved to support certain classes of object-oriented applications, and each approach has been therefore affected by the requirements of the class of applications it supports. Next, we provide an overview of the three types of objectpersistence systems.

Gateway-based object persistence (GOP) is used to support an **object**-oriented **programming** model for applications while using traditional non-object-oriented data stores to store data for an object. GOP is commonly used in cases where users want to write applications on top of existing non-object-oriented data stores using **object**-oriented **programming** models. The **objects** for an application have a different data model from that of the data store schema that is used to store the persistent state of the objects in the data store. GOP systems therefore have to perform a mapping between the object-oriented schema for the application and the non-object-oriented data store schema used to store the data. At run time, these systems translate objects from the representation used in the data store to the representation used in the application and vice versa. For ease of use, GOP systems make this translation process transparent to the application programmer (except during the mapping process when user input may be needed for other than simple mappings). Gateway-based object persistence is used by several systems including VisualAge C++ Data Access Builder*, SMRC,8 ObjectStore Gateway**, Persistence**, UniSQL/M**, Gemstone/Gateway**, and Subtleware/SQL**. This approach is essentially a middleware approach, being both application and data independent.

The standards activity relevant to GOP is being developed by the Object Management Group (OMG). OMG is a consortium formed in 1989 to focus on adopting de facto standards on distributed objects. The most important specification it has adopted is CORBA (Common Object Request Broker Architecture), which defines the fundamental architecture for object interactions. In addition to CORBA, the Object Management Architecture (OMA) consists of the following additional components: CORBA services,9 CORBA facilities, CORBA domain objects, and application objects. OMG is adopting specifications on all the components except application objects. Aside from CORBA, the following adopted specifications are directly related to object persistence: Persistent Object Service, Object Query Service, Object Relationships Service, Object Transaction Service, and Object Security Service.

Object-relational DBMSs (ORDBMSs) are built on the premise that extending the relational model is the best way to meet the challenge of new object-oriented applications. As shown by relational DBMSs the relational model has been extremely successful in practice and the SQL is already a global standard. Object-relational DBMSs therefore add support for object-oriented data modeling by extending both the relational data model and the query language while keeping the already successful technology (especially the SQL) of a relational DBMS relatively intact. There are two classes of object-relational DBMSs in the market; those that have been built from scratch (e.g., Illustra**, UniSQL**), and those that are (or will be) built by extending existing relational DBMSs (e.g.: DB2, Informix, Oracle, and Sybase). This approach is essentially a bottom-up approach, being data (or database) centric.

As might be expected, the standards activity on this area is based on an extension of the SQL standard. X3H2 (the American committee responsible for the specification of the SQL standard) has been working on object extensions to SQL since 1991. These extensions have become part of the new draft of the SQL standard named SQL3. The SQL3 standard10 is an ongoing attempt to standardize extensions to the relational model and query language.

Object-oriented DBMSs (OODBMSs) are basically built on the principle that the best way to add persistence to objects is to make objects persistent that are used in an **object**-oriented **programming** language (OOPL) like C+ + or Smalltalk. Because OODBMSs have their roots in **object**-oriented **programming** languages, they are frequently referred to as persistent **programming** language systems. **Object**-oriented DBMSs, however, go much beyond simply adding persistence to any one **object**-oriented **programming** language. This is because, historically, many object-oriented DBMSs were built to serve the market for computer-aided design/computer-aided manufacturing (CAD/CAM) applications in which features like fast navigational access, versions, and long transactions are extremely important. Object-oriented DBMSs, therefore, support advanced object-oriented database applications with features like support for persistent **objects** from more than one **programming** language, distribution of data, advanced transaction models, versions, schema evolution, and dynamic generation of new types. Even though many of these features have little to do with object orientation, object-oriented DBMSs emphasize them in their systems and applications. There are several object-oriented DBMSs in the market (e.g., Gemstone**, Objectivity/DB**, ObjectStore**, Ontos**, 02**, Itasca**, Matisse**). This approach is essentially a top-down approach, being application (or programming language) centric.

A standard for OODBMSs has been specified by the Object Database Management Group (ODMG). ODMG is a consortium that consists mainly of OODBMS vendors. ODMG has specified the ODMG-93 standard, published in book form." ODMG-93 defines an Object Definition Language (ODL), an Object Query Language (OQL), and C+ + and Smalltalk language mappings to ODL and OQL. ODMG is currently working on Java* * language mappings to ODL and OQL.

In the remainder of this paper, we describe the major characteristics and requirements of object-oriented applications and how they affect the choice of each of the three approaches to object persistence previously mentioned. We subdivide the discussion into three sections: data modeling, data access, and data sharing. In the data modeling section, we focus on the various programming language features used in building object-oriented applications. Next, in the section on data access, we focus on mechanisms for creating and accessing persistent data. Finally, in the data sharing section, we focus on shared access to persistent data. In each of the three sections, we consider various specific features, and compare and contrast the three approaches to object persistence (GoP, ORDBMS, and OODBMS). In order to simplify the understanding of these issues, we have provided a summary of our discussion in tables that appear in each of the sections.

Data modeling

Object-oriented applications that are programmed in existing **object**-oriented **programming** languages like C+ + and Smalltalk use a number of object-oriented modeling features like encapsulation, inheritance, and dynamic binding. The reader is assumed to be familiar with all the features used in developing applications using an **object**-oriented **programming** language like C+ + or Smalltalk (for instance, refer to Reference 12 for C+ + application development techniques). While object-oriented DBMSs might not support all of the features available in a native **object**-oriented **programming** language, the data models they support are much more complex than the data models supported by a traditional relational DBMS. Object-relational DBMSs are now beginning to support many of these features.

One of the main problems that object-oriented DBMSs solve by supporting the data model of an **object**-oriented **programming** language is the impedance mismatch 3 problem that exists in relational DBMSs, where the data model used in the application is different from that of the data model used in the database. This difference in data models causes two major problems for applications thus resulting in the impedance mismatch:

1. An application programmer has to program in two different languages with distinct syntax, semantics, and type systems, namely, the application programming language (e.g., C++ or oo COBOL) and the data manipulation language of the DBMS (i.e., SQL). The logic of the application is implemented using the programming language while SQL is used to create and manipulate the data in the database.

2. When any data are retrieved from a relational database, they have to be translated from their database representation to the in-memory programming language specific representation for the application. Similarly, any updates needed to be made to the data have to be explicitly communicated to the database using another SQL statement, causing the data to be translated from the in-memory representation back to the database representation. All this communication back and forth between the database and the application leads to unnecessary processing that could be entirely eliminated if the same data model were used in both the application and the database.

Object-oriented DBMSs (OODBMSs) avoid the impedance mismatch described above by providing extensive support for the data modeling features of one or more **object**-oriented **programming** languages. In an OODBMS, therefore, the data model that is used by an application is identical to the data model used by the DBMS to store the application data. OODBMSs have had great success in solving the second problem mentioned above, but they are less successful in solving the first problem, especially when object query is involved.

Object-relational DBMSs (ORDBMSs) also start to address impedance mismatch by providing more and more support for the data modeling features of major **object**-oriented **programming** languages. However, the data model that is used by an application can be close but not identical to the data model used by the DBMS to store the application data. Therefore, ORDBMSs will not be as successful in solving the second problem mentioned above, but they can be as good in solving the first problem, especially when object query is involved. ORDBMSs mitigate the second problem by providing rich support to execute portions of the application within the database server. Such support is typically provided by extending the relational support for stored procedures to now support language environments that can in turn execute user defined functions and methods.

The gateway-based object persistence (GOP) attempts to alleviate impedance mismatch through the use of schema mapping tools and automatic code generation. The intent is to give the user the illusion of working with only one data model-the data model used in the application. (The one exception is the user who defines the mapping between the data model used in the application and the data model used in the DBMS.) Therefore, it appears to the user as if an object-oriented DBMS is being used, thus solving the first problem mentioned above in a similar fashion. The GOP additionally provides facilities to automate and optimize whatever conversions are required, thus alleviating the second problem.

As might be expected, several complex issues arise in providing support for an object-oriented data model. We now proceed to discuss these issues in more detail (See Table 1).

Object identity. Object identity is one of the most important issues that needs to be handled for **object** persistence. In a **program** running as a process, **objects** can be created, copied, deleted, and accessed. Since none of these transient objects persists beyond the life of the process, the virtual memory address of an object in a process can be used as the identification (ID) of the object (OLD). In a DBMS, OIDs, like data, have to be persistent. An OID by definition refers to exactly one object in the database. The reference to the same OID for an object by an application and by another object in the DBMS refer to the same identical object.

Non-object-oriented DBMSs also have to wrestle with the problem of object identity (or record identity) but they are usually able to get by with value-based identity. Relational DBMSs typically support valuebased access to persistent data, i.e., if an application needs to access a particular row in a database, it has to query the database using the name of the relation that the row is in and a primary key value that is equal to the value of the primary key value of a row in the table. This form of access to persistent data alone is inadequate in an object-oriented application, since objects might actually have identical values but be different objects. This is because object-oriented applications support non-first-normal-form values where an object can contain another object (e.g., two employees might own the same make, model, and year of a car but each respective car object might not be shared between the two employees, resulting in an identical valued car object in each employee object). OIDs might also be needed for direct access to an object in a database.

In a GOP environment, where distributed and heterogeneous systems prevail, it is difficult to expect or require uniform OID representations. Object identity support in a GOP system will be limited by the database or file system

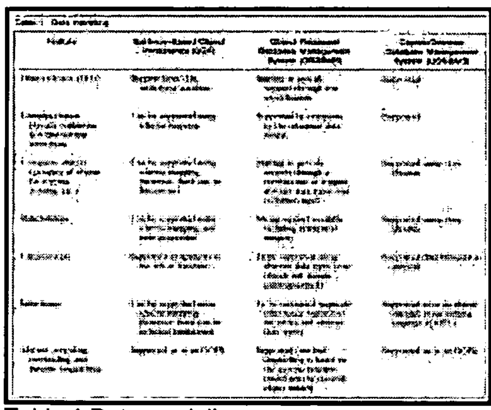
(e.g., relational, network, flat file) that stores the underlying data.

Some ORDBMSs are also beginning to provide an OID, as well as the traditional value-based object identity. One method for supporting OIDs in an ORDBMS is by creating an ID for every row in the database independent of the values in the row. Every row in any table of an object-relational database can then be directly accessed using the ID for the row.

The best support for OIDs is found in OODBMSs, and all OODBMSs implement some form of OIDs. In the ObjectStore DBMS,14 for example, database references can be thought of as equivalent to OIDs. The application merely has to provide the reference, and the database in which the reference resides is automatically opened and the object retrieved. It is possible, however, that the object does not exist anymore and retrieving an object using a database reference in ObjectStore could result in an error. Other OODBMSs also provide support for OIDs in a similar though not identical manner.

Complex objects. A complex object mechanism allows an object to contain attributes that can themselves be objects. In other words, the schema of an object is not in first-normal-form, unlike relational tuples whose schema is in first-normal-form (i.e., their components, or columns, can only be simple base types like integer, character, or BLOB). Examples of attributes that can comprise a complex object include lists, bags, and embedded objects. An example of a complex object definition in C+ +-like syntax is:

class Person char *name; int age; Car car;



Enlarge 200%

Enlarge 400%

Table 1 Data modeling

Set(Person) children; List(string) phones; Set(Person) same age;

In the above example, an instance of the class Person is a complex object that contains as attributes two basic attributes (name and age), an embedded Car object (car), a set of Person objects (children), a list of character strings (phones), and another set of Person objects (same-age), where all the objects are embedded.

Complex object support (previously mentioned) is extremely useful in modeling non-first-normal-form schemas that occur routinely in most object-oriented applications. The objects that are defined inside other objects (e.g., the attribute car inside class Person) are entirely part of the containing object and do not have any identity of their own-components of complex objects are automatically created (recursively) when the top-level object is created and are automatically deleted when the containing object is deleted. The connection between the complex object and its component is always valid and cannot be removed. In this respect, complex objects differ from the composite objects that are described next.

In GOP systems, complex objects in the data model of an application need to be mapped to the underlying data in a data store. Any mapping is typically accompanied by some application-specific generated code that can translate back and forth between the data model in the application and that used in a database. Such mappings can be quite inefficient if the underlying database is not equipped to store the complex object. For example, consider the following two solutions for storing a (fixed length) array attribute for a complex object in a relational DBMS:

I. Store the elements of the array in a tuple with one column for each element of the array (multiple columns will be

needed per element if the array element itself is a complex object).

2. Store the array element in a separate table with every tuple of this table storing a single array element with the index of this element in the array.

Obviously, neither of these solutions works very well (and does not work at all for variable length arrays). Another solution that has been proposed for this problem is to store a complex object in a BLOB (binary large object) field.

ORDBMSs provide extensions to the relational data model to support non-first-normal-form data such as lists, bags, sets, etc. These extensions in the data model can then be used to define complex objects in an application.

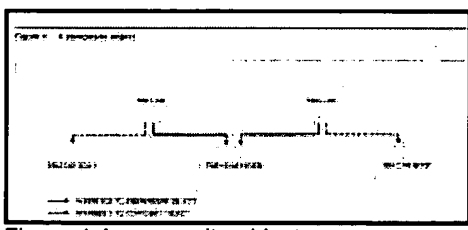
OODBMSs are extremely strong in supporting complex objects since they are based on **object**-oriented **programming** languages that have extensive features for defining complex objects in the data model.

Composite objects. Composite objects are individual objects that are related and form part of a group. Typically object-oriented applications utilize a composite object as a group of objects that are part of a parent object that is typically a collection. A composite object is an object that might have to be treated as being owned by a particular object but can be pointed to by other objects in a normal way. The pointer from an owning object to an owned object is special. An example of an application scenario where composite object support is needed is given in the next paragraph.

A design object for a car might consist of the design objects for the engine, the power train, the wheels, the body, etc. It is quite possible that the engine (and, therefore, its design object) remains unchanged for several related car models while the external body shape is different for each model. In such a case, it is necessary to be able to treat some portions of the car object and its components as one entity while sharing other component objects with design objects of other cars. An example definition of composition is shown as follows and an example scenario is illustrated in Figure 1:

class Car(char *name; char *model; Ref(Engine) engine; CompositeRef<Body) body;

Associated with composite objects is the issue of cascading the deletion and copying. In certain applications, if an assembly object is deleted, the component objects are kept around since they can be reused for alternative assemblies, e.g., a car model might go out of sale but the engine design might continue to be used in other new models. While copying objects, it might still be necessary to copy every component object in an assembly. In other words, cascading the delete might not be needed, while cascading the copy (sometimes referred to as deep copy) might be essential. Such properties can be specified using properties of the composite object.



Enlarge 200%
Enlarge 400%

Figure 1 A composite object

In GOP systems, composite object support is provided using a combination of schema mapping and application-specific code generation. However, since GOP systems depend on other autonomous database systems to store the data, there might be limitations to the support that can be provided (e.g., clustering at an object level might not be available at the data store and therefore cannot be provided for composite objects).

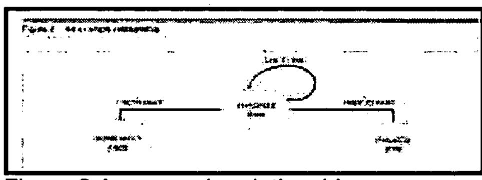
ORDBMSs are beginning to provide support for composite objects using a combination of triggers (used for propagation of delete, for example), abstract data types (ADTs), and collection types (e.g., lists, bags, sets, etc.). While propagation of deletion and copying can be easily supported, it is not clear how ORDBMs will support clustering of components of a composite object near each other. This is because ORDBMSs, by definition, cluster at the table level and not all of them can cluster rows from different tables on the same disk page.

Composite object support might or might not be explicitly supported by the various OODBMSs. In ObjectStore, for

example, composite object support is not explicitly present. Nevertheless, an application can implement complex object support by using relationships with the property that the deletion (or copying) is propagated. In the Versant** OODBMS the extent of a class can be thought of as a composite object that collects all of the instances of the class. Deleting an object will automatically remove it from the extent of its class. Deleting an extent for a class will result in deletion of all instances of the class.

Relationships. Relationships are a generalization of referential integrity constraints in relational DBMSs where a particular foreign key points to the primary key, and this reference is automatically maintained by the database. Relationships in a DBMS are references between objects in a database and have the following features: automatic propagation of deletion, setting one side of a bidirectional relationship automatically sets the other side also, and deleting an entry from one side also automatically deletes the inverse entry on the other side (i.e., referential integrity is maintained).

In Figure 2, we show an example of three types of relationships that occur commonly in applications. First, employees (represented by the Emp class) can be related to other employees via the spouse relationship that is an example of a one-to-one relationship. In addition, employees are related to departments (represented by the Dept class) by a one-tomany relationship, i.e., every employee can work in exactly one department while a department can have many employees. Finally, employees work in projects (represented by the Proj class), an example of a manyto-many relationship, i.e., an employee can work on many projects and a project can have several employees.



Enlarge 200%

Enlarge 400%

Figure 2 An example relationship

In a GOP system, relationships can be supported at the application level by a combination of schema mapping and appropriate query generation at run time to automatically retrieve related objects. Relationships at the application level need to be mapped into the database using the database features available, such as primary keys, foreign keys, and row ids. This may pose some limitations, especially in terms of performance, since retrieving related objects might require multiple join queries to be executed against a traditional relational DBMS. The OMG Relationships Service specification also describes relationships spanning more than two object types resulting in special objects to implement relationships.

Object relational systems provide extremely strong support for relationships. This is to be expected since traditionally, their precursors, relational DBMSs, have provided excellent support for referential integrity. Row ids and collections in an ORDBMS, along with the referential integrity support, can be used to fully support relationships.

Since OODBMSs allow lists and sets as attributes inside an object, these relationships can be implemented between objects with embedded sets of pointers to store the associations. Typically OODBMSs support two-way relationships whose members are maintained using embedded sets in the objects being related (no new objects are needed to implement a relationship). Next we illustrate how such a schema might be defined using the ODL (Object Definition Language) of the ODMG standard. " Individual OODBMSs use variations of such a syntax to define such a schema. class Emp Ref(Dept) dept inverse emps_in_dept; Liat (Ref(Proj> projects inverse emps_in_proj; Ref(Emp) spouse

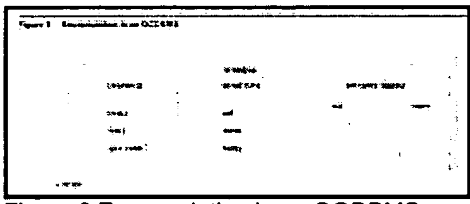
}

class Dept { Set(Ref(Emp> emps_in dept inverse dept;

} class Proj { Set(Ref,emp..emps_in_proj inverse projects;

The non-first-normal-form support in OODBMSs makes the representations of many-to-many relationships here more compact than in an equivalent relational schema. For instance, an intermediate table would be required in a relational schema in order to model the many-to-many relationship between projects and employees.

Encapsulation. The programming language view of encapsulation clearly differentiates between the method of accessing an ADT (abstract data type), referred to as the interface, and the internal data structure used to implement the ADT. An ADT interface is available to users of the ADT and does not change with any change in the ADT internal structure. The interface is therefore said to encapsulate the internal structure of the ADT (we refer to this as procedural encapsulation). In **object-oriented programming** languages, the ADT is typically also referred to as a class and the interface is referred to as a set of public methods. For example, a stack ADT might have methods push(elemtype), empty(), pop() and the stack itself might be implemented using a fixed-length array. Later, if we decide to change the implementation of the **stack** to use a more **dynamic** data structure such as a linked list, none of the applications using the stack needs to be changed (in some cases, the application needs to be recompiled).



Enlarge 200%
Enlarge 400%

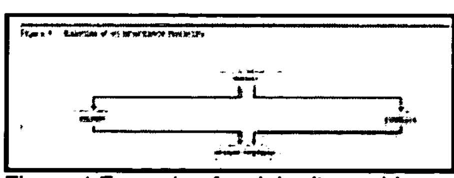
Figure 3 Encapsulation in an OODBMS

While making objects persistent, one more level of implementation needs to be considered in addition to the interface and the internal structure present for an ADT in a programming language, namely the physical database implementation. The physical database implementation determines (1) whether the data are stored in sorted order of primary key or as a heap, (2) the primary and secondary indices available to access the data, and (3) the clustering strategies (e.g., a Department object might be stored physically clustered with all of the Employee objects belonging to that Department object). In Figure 3, the class Employee has an interface with three methods, hire(), fire(), and give-raise(). The data structure of each Employee object is a record with three fields, namely eid, name, and salary. The physical database design consists of the Employee objects stored as a file sorted in order of eid indexed by two indices, one on eid (the primary index that is clustered) and another on name (a secondary index, unclustered).

Encapsulation in GOP systems is supported at the application but not in the data model used to store the data in the underlying database. Objects that are encapsulated for the application need to be constructed from the data in the database using a translation mechanism that uses the schema mapping as well as application-specific generated code libraries. This scheme is extremely flexible as different applications using the same underlying data can use different schema mappings and therefore different encapsulation rules.

ORDBMSs support encapsulation using ADTs that can be columns in a table. The row objects of a table themselves are not encapsulated. This is because object-relational systems are ("backwards") compatible with the first-normal-form relational model where columns of tables are unencapsulated. A schema for a table is used in queries using SQL and hence encapsulation does not make sense here. Interestingly, even OODBMSs break encapsulation rules for queries, as can be seen later.

OODBMSs depart from the strict procedural encapsulation of ADTs enforced by programming languages, and sometimes allow direct access to the structure of a class bypassing the methods. Breaking of encapsulation rules is usually done in OODBMSs for executing ad hoc queries. Always accessing a class by using a method might not be efficient, and sometimes even be inadequate if there is no method to get the required answer. A query might loosely be thought of as a dynamically defined procedure used to compute the answer by looking at the internal structure of one or more classes of objects. Ad hoc queries are discussed in more detail in a later section.



Enlarge 200%
Enlarge 400%

Figure 4 Example of an inheritance hierarchy

Inheritance. Object-oriented systems use inheritance in order to realize implementations that mirror real situations. For example, in a university database, the inheritance hierarchy shown in Figure 4 might be used. Inheritance,

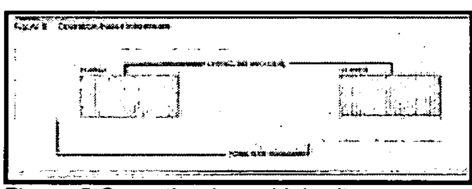
which is a powerful modeling tool, in conjunction with the encapsulation support described earlier, enables sharing of implementations across classes that are part of the same inheritance hierarchy. There are several types of inheritance in use in the various programming languages and OODBMSs are affected by these. We will describe two major types of inheritance, namely operation-based inheritance and structure-based inheritance.

In operation-based inheritance, class B is said to inherit from class A (i.e., class B is a subclass or subtype of class A), if for every (public) method in class A, there is an equivalent (public) method in class B that has an identical interface. In other words, for the purpose of method calls, an instance of class A can be replaced with an instance of class B. Operation-based inheritance is illustrated in Figure 5.

Note how the structures of class A and B are completely different, but that does not affect the fact that class B is a subclass of class A. An example of a language that supports operation-based inheritance is Smalltalk.

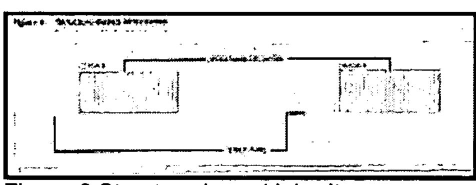
In structure-based inheritance, class B is said to inherit from class A, if for every (public) method in class A, there is an equivalent (public) method in class B that has an identical interface, and if class B has a superset of the structure of class A. In other words, an instance of class A can be replaced by an instance of class B for accessing structural information of A in addition to method calls. Structure-based inheritance is illustrated in Figure 6. Note that the structure of class B contains every aspect of the structure of class A, in addition to the methods of class A. Structure-based inheritance is more restrictive than operation-based inheritance. An example of a language that supports structure-based inheritance is C++.

Application designers who want to address multiple platforms (i.e., port) should be aware that applications that make use of structure-based inheritance are not as easily portable and extendable as those built using only operation-based inheritance. The IBM system object model (SOM), 15 which can be used for developing a single application to be run on multiple platforms, therefore, only supports operationbased inheritance. One advantage of structure-based inheritance is that objects are compact and most method calls can be dispatched at compile time. This turns out to be very efficient. Operation-based inheritance requires more work (such as method lookup and integrity checks) to be performed at run time, and hence could be slow for applications that access millions of objects and call several methods on each of these objects.



Enlarge 200%
Enlarge 400%

Figure 5 Operation-based inheritance



Enlarge 200%
Enlarge 400%

Figure 6 Structure-based inheritance

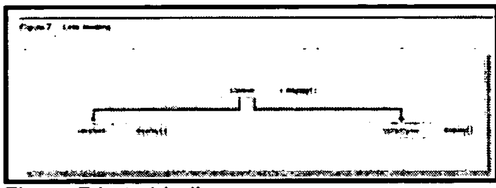
Multiple inheritance is the term used when a single class can inherit from multiple parents. In Figure 4, the class Student-Employee inherits from both Student and Employee and is an example of a multiple inherited class. Multiple inheritance is used rarely in real applications, and we will not describe this issue further except to state that OODBMSs support multiple inheritance if the underlying programming language that they use also supports it. Multiple inheritance could cause porting problems since different systems that implement this feature use incompatible rules for resolving conflicts, such as those between inherited method names.

In GOP systems, inheritance can be supported using schema mapping. The mapping, however, can become quite complicated (as in the complex object case discussed earlier). For example, the hierarchy stored in Figure 7 can be implemented in two different ways using tables:

1. There could be one table for the inheritance hierarchy consisting of Student, Employee, and Person. This table would have columns to store the combined attributes of all classes in the inheritance hierarchy. In addition to that, there would also be a special column that would tag the row with the object type in that row, namely Student,

Employee, or Person. This scheme obviously wastes a lot of space for attributes that are not needed for a particular object, but the columns need to be stored.

2. A normalized version to store the same data is in three separate tables, one each for the classes Student, Employee, and Person. All of these tables could share the primary key (i.e., the primary key for the root class table is the same key pointed to by objects in any subclass table). The tables for a subclass will only contain columns for the attributes for that subclass thus avoiding the wasted column space in case 1 above.



Enlarge 200%
Enlarge 400%

Figure 7 Late binding

Using solutions like that previously discussed to model inheritance could become quite complicated, since an application that needs to access all Person objects needs to write complex queries to figure out what objects belong to Person or any of its subclasses (since a Student or Employee object is also considered a Person object).

ORDBMSs support inheritance, but two separate hierarchies for tables and ADTs are used. The row objects of a table are unencapsulated while the ATs of a column can be encapsulated. The functionality and semantics of these two types of inheritance are evolving with continuing work on the SQL3 standard. For example, it is not clear what it means to have inheritance for tables since tables are unencapsulated.

OODBMSs support the inheritance features in already existing **object**-oriented **programming** languages and, unlike SQL3, do not invent a new semantics on their own. However, in order to support cross-language support for objects stored in the database (i.e., access a Smalltalk **object** from a C+ + **program**) they impose some limitations on the features of the programming language that can be used in developing such applications.

Method overriding, overloading, and dynamic binding. Inheritance allows users of a class hierarchy to extend the interface of the hierarchy by redefining methods that are already defined at higher levels in the class hierarchy. Such overriding of a method allows a method that is declared at a superclass to be reimplemented for a subclass. In addition, objectoriented programming systems allow overloaded methods where a new method can be defined with the same name as an already existing method but with a different set of arguments (also known as a signature). Both overriding and overloading are very useful for writing easy-to-read code, but combined with support for late binding (where method resolution is deferred to run time) also make it useful to write applications that can make use of any future enhancements to existing class libraries used by an application. We will illustrate this using an example shown in Figure 7. The base class Person has its own implementation of display() and so do the derived classes Student and Employee. Due to the properties of inheritance, a reference to a Student or an Employee object can also be stored in the set teenagers. It is now possible to write generic code to display a set called teenagers that is a set of references to person objects without the code knowing the subtype of the individual types of objects that are being displayed:

Set(Ref(Person)) teenagers; Ref(Person) person;

for person in teenagers do person.display(); In GOP systems, the **objects** at the application are **programming** language **objects**. These methods are nonexistent at the database and run only using the object representation of the data. The data from the database have to be translated to the object representation and the application code is run just like an in-memory program. Methods are therefore dispatched using the mechanisms in the programming language.

ORDBMSs store both methods and data within the database and they are able to dispatch methods on objects (ADrs) within the DBMS server. Method dispatching is based on the generic function model (calling a function with the object as its first argument) rather than the classical object model (which is based on sending a message to the object). Methods can also be used in queries, stored procedures, and userdefined functions, as well as in application programs. Executing methods at the server can be quite challenging since there is no access to the user terminal from within the DBMS server. However, there have recently been instances of stored procedures being able to output hypertext markup language (HTML) statements for displaying on a browser connected to the

World Wide Web portion of the Internet. OODBMSs typically execute methods at the client since these methods are written in a programming language whose environment is available only at the client. Most systems do not store methods in the database.

Data access

Having discussed the various data modeling features and associated issues, we next explain how application objects can be created and stored and discuss the support provided for navigational and ad hoc query types of access to persistent data. In discussing these, we briefly mention the interaction between client and server, particularly the method by which objects are communicated between client and server.

Finally, we discuss some important application support items including schema evolution, integrity constraints, and triggers. Table 2 summarizes this discussion.

Creating and accessing persistent data. The best way to support persistence is to do it in a way that is orthogonal to type (i.e., it is possible to create persistent and transient objects of the same type in an application). Typically, there are two main ways (sometimes both are supported in the same system) of adding persistence to objects of an instance, either by overloading the new operator, or by requiring that every class having persistent instances inherit from a common class whose definition and implementation is provided by the database system.

In the operator-based approach, an overloaded global new operator is used to create a class instance. Applications that need to create persistent instances of a class will have to create objects using the systemprovided new operator. Example calls to create new objects using the operator-based approach look like:

Ref(Employee) temp-emp = new Employee;

// The above is a transient employee object Ref(Employee) pers emp = new (myDB) Employee;

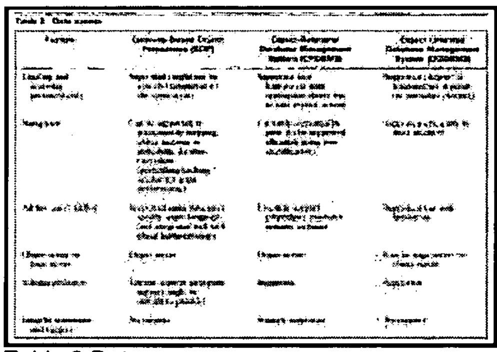
The first statement above (along with its comment line) creates a transient object, while the second one creates a persistent object in the database myDB. The operator-based approach has the advantage that the class definitions of the application remain unchanged and hence all facilities available in the programming language can be used as they exist. It is possible that existing applications for manipulating transient data can be migrated to access persistent data with relative ease using this approach. The disadvantage here is that the schema information must be somehow transmitted to the OODBMS by translating from the programming language version. OODBMS products like ObjectStore and 02 use this approach for creating persistent data.

Inheriting from a common class is another popular way of providing persistence. Many OODBMSs such as Objectivity and Versant use this approach. GOP systems such as VisualAge C+ + Data Access Builder also use this approach. In this approach, an object is made persistent by the type of the object-in other words, persistence is not orthogonal to type. Inheriting from a common class has the advantage that the schema information may be deduced from the result of calling derived and redefined methods. This in turn implies that certain methods have to be implemented in every class that needs to contain persistent instances but this is mitigated by the fact that existing OODBMSs provide automatic generation of application class definition files with the code for the inherited methods that are generated automatically. The application designer needs only to implement the application methods for the class, since the database-specific methods are generated automatically. Unfortunately, however, this scheme almost always results in multiple inheritance needed by the application in order to merge the inheritance hierarchy for the system with that of the application. Since multiple inheritance is not supported very well in many **object**-oriented **programming** systems (it is rarely used in practice), the result is a more complex and less portable application.

Reading persistent data can be made virtually transparent to the application in all three types of systems. Updating data, however, is another issue. In a GOP system, updates are typically not transparent and an application will need to inform the system explicitly of objects that have been changed (some encapsulation is possible here, for example, update of relationships, but changing an atomic field like an integer is impossible to encapsulate). In an ORDBMS, updates are done using a separate UPDATE statement and therefore are nontransparent. Finally, OODBMSs vary in their degree of transparency, ranging from ObjectStore where updates can be made completely transparent, to other systems such as Versant where an object has to be explicitly marked "dirty" by an application. In fact, the ODMG-93 standard has a special interface defined for marking dirty objects from an application.

Navigation. As mentioned at the beginning of this paper, OODBMS development was driven by the applications that needed fast navigational access (e.g., verification and routing an integrated circuit might be an extremely CPU-intensive operation that requires fast access to component objects and other component objects). Some of these OODBMSs (e.g., ObjectStore) provide extremely fast navigational access to data by making use of operating system support for page faulting. J Typically, the first access to a data item in the database results in the item being swizzled (resolution of a page fault condition) to an in-memory representation and subsequent access to the memory location is extremely fast. In some cases such as ObjectStore, the data stored in the OODBMS are identical in size and structure as the in-memory structure and hence after the first access, subsequent accesses to a data item are as fast as in an in-memory program. This excellent performance comes at the cost of a tight integration of the application code with the database client code that results in reduced security. Fast navigational performance therefore comes at a cost and application developers must be aware of these trade-offs when using an OODBMS.

Navigation using a GOP system can be supported by mapping object accesses to underlying accesses to the databases that store the data. Naive algorithms for navigation using a relational database could cause very poor performance (generating one SQL query for every object access). GOP systems tackle this performance problem using a two-fold strategy, (1) by maintaining a large cache of application objects in main memory, and (2) by providing facilities for fetching (prefetching) objects before they are needed. Such prefetching usually needs to be specified by the application, thus ensuring that all of the objects needed can be prefetched using one query.



Enlarge 200%
Enlarge 400%

Table 2 Data access

Ad hoc query facility. Relational DBMSs have been tremendously successful mainly due to the ad hoc query language that they support (the SQL standard).7 As we mentioned earlier, the SQL is different from the programming languages used by applications and this causes an impedance mismatch between the query and the applications.

A GOP system typically does not implement a new query language on the object representation. Applications using GOP, therefore, tend to use the underlying data-store-specific query language (almost always SQL) for executing queries. The query works on the underlying data model that is not object-oriented and this does not work well with the application object model. In other words, the application is left with a worse impedance mismatch here than exists in traditional relational DBMS systems.

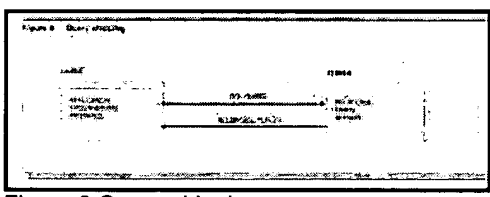
An ORDBMS has excellent support for queries and handles most of the work in optimization and index management very well. Unfortunately, some impedance mismatch still could remain since the application data model might still be different from the database data model. By moving as much of the application into the ORDBMS using ADTs, user-defined functions, and methods, applications can solve this mismatch to a large degree. The problem, however, is that with client machines having enormous caches and computing power, this server-centric approach might not be cost effective and might even be unsuitable for highly interactive applications.

The query language supported by an oODBMS is an extension of the **object**-oriented **programming** language for which it is designed. Therefore, OODBMSs mitigate the impedance mismatch by using the same type system for ad hoc query and programming methods. Typically, OODBMS query languages do not obey encapsulation rules and are allowed to access the structure of the data. This is unavoidable since ad hoc queries by nature require arbitrary computations on the data that cannot be captured a priori using a fixed set of methods. Unlike ORDBMSs that support the dynamic creation of views, OODBMSs typically have no support for dynamic view creation. Derived attributes can, however, be implemented in a more static manner using the inheritance hierarchy and late binding. The support for queries varies significantly between the various commercial OODBMS products. Some of them until

recently provided no query language, or provided support for queries with no support for query optimization and index management, thus making query support virtually useless in practice. Other OODBMSs such as ObjectStore restrict queries to semijoins as opposed to arbitrary relational expressions supported by SQL-ObjectStore queries can only start on one root collection and the result generated cannot generate new types dynamically. ObjectStore, however, does provide support for index management, automatic index maintenance in the presence of updates, and query optimization. Finally, OODBMSs such as 02 support an unrestricted and powerful query language with query optimization and index management. Of late, the query languages provided by the various OODB products are beginning to converge to the ODMG93-specified object query language (OQL) query standard, which unfortunately does not seem to be fully compatible with the SQL standard.

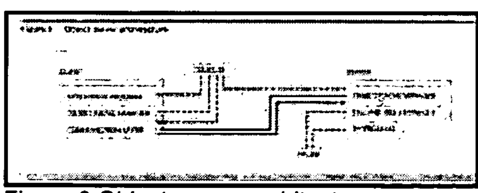
Object server versus page server. In a client/server architecture, there is a division of labor between the client and the server, and database management systems need to make use of the resources available at the client and the server efficiently. A relational DBMS client accesses data from the server using a mechanism known as query shipping, shown in Figure 8. In query shipping, the client sends SQL queries to the server. The server, on receiving the query, optimizes the query, picks a suitable access plan making use of any available indexes, and executes the query. The result of the query is a set of relational tuples that are returned to the client. The client then processes the data and initiates further queries if necessary. The tuples returned from the server to the client are first-normal-form relations except for large objects (e.g., CLOB, BLOB).

The availability of relatively inexpensive workstations with powerful processors and large amounts of memory has resulted in driving computer systems from a server-centric model to a more balanced workload distribution between the client and server. OODBMSs have been affected by this trend and make a different trade-off between the client and the server resources than the relational DBMS systems do. The typical OODBMS architecture results in more work being offloaded from the server to the client than a typical relational DBMS. In many OODBMSs (e.g., ObjectStore), all but the essential jobs of storage management, concurrency, and recovery is offloaded to the client. The data (including those in secondary access structures like indexes) are shipped to the client that possesses all the intelligence to execute queries, methods, etc. OODBMSs, therefore, perform data shipping as opposed to the query shipping done in relational DBMSs. Data shipped between the client and the server can be either objects (i.e., the server provides objects to the client) or pages (where the server provides pages to the client that contain objects without knowing what objects are contained in the page). We will now explain the page server and object server architectures in some more detail.



Enlarge 200%
Enlarge 400%

Figure 8 Query shipping



Enlarge 200%
Enlarge 400%

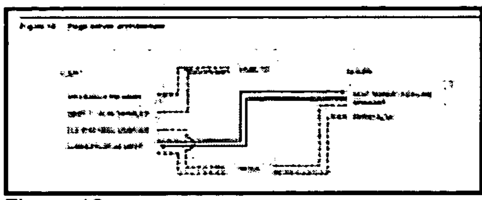
Figure 9 Object server architecture

The object server architecture is illustrated in Figure 9. An object server can either receive requests for a single object (using, for instance, an object identifier) or a set of objects using a query. As shown in

Figure 9, the communication between the client and the server is based on objects, and object caches can be maintained on both the client and the server, thus making use of the availability of inexpensive and abundant memory. The objects themselves are eventually stored on a disk clustered physically into pages but this is completely managed at the server with the client being virtually unaware of this representation.

The object server architecture has the advantage that queries, and more importantly methods, can be run on both the client and the server. This means that the communication between the server and the client can be minimized by using auxiliary data structures such as indices to evaluate queries at the server. Locking and constraint management are simplified since these critical tasks can be performed at the server. Robust authorization and

security of the sort implemented in relational DBMSs can be implemented in an object server architecture-in fact, a relational DBMS can be thought of as an object server that serves primitive objects (first-normal-form tuples).



Enlarge 200%
Enlarge 400%

Figure 10

In contrast, however, object servers have many disadvantages making them hard to implement. For instance, query processing is complicated in the presence of updates to objects-either the changed data on a client have to be flushed to the server before executing any query on the server, or queries have to be executed at both the client and the server and the results merged. The former strategy results in poor performance and makes caching virtually useless in the presence of updates, while the latter strategy results in all of the hard problems associated with distributed-query processing. When objects exist on both the server and the client, there is a practical problem related to the **object**-oriented **programming** language used to implement method code. Virtually all of the code written in **object**-oriented **programming** languages needs a run-time environment to execute it. Running methods at the server means that the method code has to be stored in the DBMS and the DBMS has to understand the run-time requirements of all of the **programming** languages whose **objects** are stored in the database. Finally, since objects are stored on pages in a disk at the server, an object server has to pack and unpack these objects for transferring them to and from the clients. This could result in more work at the server resulting in bottlenecks (see Reference 13 for a study on three alternative client/server strategies for OODBMSS). Performance of an object server might be affected by having to ship complete objects across the interface, which could result in very poor performance for large objects.

An alternative to the object server architecture is a page server architecture (shown in Figure 10) where the unit of transfer between a client and a server is pages. The server in a page server architecture performs minimal functions like concurrency, recovery, and storing pages. Objects exist only at the client. On comparing the client of the page server with that of an object server (Figure 9), it is clearly seen that a server in the page server has much less functionality than a server in an object server and hence is much simpler to implement.

As might be expected, a page server simplifies the implementation of the DBMS by keeping the server function low, and good performance can be achieved by a suitable clustering of objects to pages. A page server minimizes the load on the server and such a system architecture therefore enables a server to handle more clients than otherwise before becoming saturated.

A page server has some disadvantages in that clients can become rather large in size since all database functionality has to be present at the client. Messages for accessing small objects might be larger in size than necessary (since a page is the smallest unit of transfer). More data than necessary might be transferred from the server to the client to compute the result of queries since the queries cannot be shipped to the server for remote execution. There are further implementation complexities to providing support for object locking in a page server. In spite of the above disadvantages, many OODBMSs use page servers since the access characteristics for the applications in this market can be best served using a page server. The impedance mismatch that exists using an object server makes it unsuitable for this class of applications. For a more detailed comparison of the various client/server architectures, see Reference 13.

GOP systems and ORDBMSs can be considered as object servers, but OODBMSs carl be both object and page servers. Examples of page server architectures include ObjectStore and 02. An example of an object server architecture is Versant (which uses prefetching heavilyin order to minimize the overhead of transferring many small objects between the server and the client).

Schema evolution. All relational DBMSs support schema evolution. Schema evolution involves two relatively separate parts, the first involves changing the schema, and the second involves evolving existing data that are in the form of the old schema to their new representation based on the modified schema. Since relational DBMSs have a simple data model (no new user-defined types are allowed) and since the complexity of schema evolution is directly proportional to the complexity of the schema, schema evolution support is relatively simple to provide in a relational DBMS. Typically, schema evolution can be done dynamically on a relational DBMS by changing the

catalog definitions of relations by using special operations that can be executed in the same way as an SQL command (note that special authorization might be needed for modifying the catalog). We shall see that when we move to object-oriented schemas, this support becomes somewhat more complex to provide and also harder to use for an application programmer or user.

In a GOP system, schema evolution support might be extremely limited, however, schema mapping evolution (without change in the underlying data) might be easy to achieve. In other words, the same underlying data can easily be viewed using a different object model by just creating a new schema mapping and its accompanying application-specific generated code.

ORDBMs can be expected to provide strong support for schema evolution of table definitions. Evolving complex types like ADTs and collections could cause some problems, needing user intervention to migrate old objects to their new representation.

In OODBMSs, the data model is complex (typically, it has the same features as the data model of an **object**-oriented **programming** language like C+ + or Smalltalk). We will not go into great detail on schema evolution support except to point out that since the type system of an **object**-oriented **programming** language is complex, schema evolution in an OODBMS cannot be completely automated as in a relational DBMS. In OODBMSs, user intervention might be needed to evolve objects belonging to a user-defined type from the old representation to a new one. For a detailed discussion of the issues involved in schema evolution support for OODBMSs, see Reference 16.

Integrity constraints and triggers. As stated earlier, the interface (see Figure 8) that separates the data and data model at the relational DBMS server from the data and data model of the application causes an impedance mismatch. However, this strict separation between client and server has advantages like the strict protection of security and integrity and gives the DBMS full control of its data. Relational DBMSs use this simple client/server interface along with views to implement a robust authorization and security mechanism.

In order to minimize the crossing of the high-cost interface between the client and server, relational DBMSs provide stored procedures to perform complex computations including multiple SQL statements within the DBMS. Stored procedures are written by "trusted" application developers and are executed by the DBMS at the server in order to maximize performance. Another class of support provided by relational DBMSs includes automatic execution of triggers and integrity constraints. Triggers, as their name indicates, are automatically triggered by updates to the database and can call stored procedures to perform complex tasks automatically within the DBMS. Integrity constraints are constraints that maintain consistency between foreign keys and the data that the foreign keys point to-integrity constraints can therefore be thought of as a special case of the more general triggers.

We are not aware of any GOP systems that provide support for integrity constraints and triggers. ORDBMSs, being extensions to relational systems, provide excellent support for integrity constraints and triggers. OODBMSs provide virtually no support for integrity constraints and triggers.

Data sharing

In this section we describe the support provided for applications by the various DBMSs for sharing data between concurrent users, crash recovery, advanced transaction models (long transactions, versioning, nested transactions), and distributed access to data (see Table 3).

[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]

Enlarge 200%
Enlarge 400%

Table 3 Data sharing

ACID transactions. Support for ACID (atomicity, consistency, isolation, and durability) transactions in a GOP

system might be limited since the object cache maintained at the application is loosely coupled to the DBMS (for example, it might not be possible to use two-phase commit between the application and the DBMS). Therefore, unless all data in the cache are invalidated at the end of each transaction, consistency cannot be achieved. In contrast a DBMS maintained cache would only require the log to be flushed to the database at the end of a transaction; the cache at a client continues to be valid. Locking in a GOP system depends on what is supported by the underlying database.

ORDBMSs should continue to leverage the extremely high transaction rates achieved by relational DBMS systems for "business" transactions. ORDBMSs should support all the traditional lock types available in relational DBMS (tuple, page, and table locks). In addition many relational DBMS systems also use extremely sophisticated locking techniques on indexes to avoid two-phase locking on indexes that might cause a bottleneck due to false data sharing.

OODBMSs also support the conventional type of short transactions termed ACID transactions. 17 The transaction rates supported by the OODBMSs do not yet approach the high rates achieved by relational DBMSs on standard transaction processing benchmarks. OODBMSs do support various types of locking. The standard lock types are page locks and object locks (also known as record locks in RDBMs). Locks on composite objects are supported in some systems (e.g., ITASCA), as well as a special lock that locks the extent of a class. (All instances of the class are locked -this is analogous to the table lock in a relational DBMS.) In addition, OODBMSs also support advanced types of locks such as notify locks, where a holder of the notify lock is notified if another transaction locks the item in a conflicting mode. These new types of locking modes are made necessary by the new types of applications that are typically supported by OODBMSs.

Locking performance is affected by the granularity of the lock (e.g., page versus object or tuple) and how it relates to the granularity of the data transfer between the client and the server (page versus object). Typically, implementation is simpler and performance is better if the locking granularity matches the granularity of data transfer between the client and the server. For more on how locking and caching interact, refer to Reference 16.

Crash recovery. Recovery from crashes as well as more catastrophic events like media failure are wellknown characteristics of an industrial strength DBMS. GOP systems provide whatever support is available in the underlying data store. ORDBMSs, by virtue of being extensions of the highly robust relational DBMS, will be extremely strong in this area. OODBMSs provide recovery support but this support has not yet reached the robustness found in commercial relational DBMSs (which provide more advanced features such as media recovery).

Advanced transaction models. One of the major support items provided by OODBMSs that is not supported very well by existing relational DBMSs (or GOP or ORDBMSs) is support for advanced transaction models. A model for CAD transactions was described first in Reference 18 and a lot of subsequent work has been done in that area. In these systems, applications access large amounts of data (e.g., a VLSI, or very large scale integration design database), and tasks take a long time to complete (e.g., it might take a week to design a register). Short-term locking of the type usually done for ACID transactions is insufficient since the database will be inaccessible to other users for long periods of time. In addition, logs might be filled up during the running time of long-running tasks and, typically, DBMSs need to quiesce the system (eliminate any transactions) to take care of such situations. Furthermore, in these advanced applications, rolling back a task (resetting to before the task executed) might be a normal operation.

Typically, OODBMSs handle such long-running tasks by providing support for versioning objects. Typically, objects can be versioned, versions built into configurations, and a concept of work space is implemented to access the latest version in a configuration. Versioning, by itself, is a very complex topic and the reader is directed to Reference 19 for a detailed look at versioning issues in a CAD system.

Object-oriented systems provide various levels of support for versioning. All of these systems provide support for versioning at the object level, some of them at the composite object level (e.g., ObjectStore, Objectivity, Versant), and allow access to a specific version or even multiple versions in case merging two versions is needed. In some cases, entire configuration support is provided (e.g., ObjectStore), and in other cases, primitives are supported to enable users to build configuration support (e.g., Objectivity). Most systems support both static access to a particular version of an object (e.g., version 1.0 of the source code) as well as dynamic access to a current or latest version of the object (e.g., the latest released version of a product).

Security, views, and integrity. The traditional relational DBMS support for security is extremely strong.

In particular, relational DBMSs (and, therefore, ORDBMs) support robust security mechanisms using the view mechanism, and by ensuring that the entire application executes in its own address space apart from the DBMS server address space (except for stored procedures that execute in the server address space for performance reasons). In contrast, OODBMSs, by using the page server concept, allow clients to cache data for acceptable performance. They do not typically have support views at all and the protection is at a coarse level of granularity (typically, at the page or even segment level).

Conclusions

This paper has discussed in some detail the features used in object-oriented applications and how well these features are supported in the three classes of object persistence systems. In our discussion, we have classified the systems into three categories:

1. The gateway-based object persistence (GOP) approach, which involves adding **object**-oriented **programming** access to persistent data stored using traditional non-object-oriented data stores like relational databases, hierarchical databases, or flat files

2. The object-relational DBMS (ORDBMS) approach, which involves enhancing the extremely popular relational data model by adding object-oriented modeling features like abstract data types to it

3. The object-oriented DBMS (OODBMS) approach (also called the persistent programming language approach), which involves adding persistence support to **objects** in an **object**-oriented **programming** language like Smalltalk or C+ +

We now conclude by providing a short summary of the three approaches to object persistence and what general classes of applications are best suited to each of these approaches.

The GOP approach is a "middleware" approach, being both application- and data-independent. It is a very good approach for integrating diversified enterprise information systems and providing a common framework for building object-oriented applications. It is also extremely good for managing shared, distributed, heterogeneous, and languageneutral persistent business objects. One main advantage of building a GOP application is that legacy applications continue to work on data that are also being accessed by the new application. While GOP is extremely good for providing object-oriented access to legacy non-object-oriented data, it is not very good for storing arbitrarily complex objects in a legacy database system. As we pointed out earlier, there are some disadvantages (bad performance and complex application logic) to blindly mapping object-oriented models to non-object-oriented databases. GOP applications can, however, access other OODBMSs and can store complex objects natively in them while continuing to access and update data in legacy databases. This field is still in the formative stage and has many technical challenges lying ahead. Some critical challenges include the integration of object persistence with object query, object transaction and workflow, and object security. The OMG group is in the process of specifying standards in this area. Applications that have an overwhelming need to access legacy data and heterogeneous data access, while allowing legacy applications to continue to work on the legacy data, are best suited for using GOP systems.

The ORDBMS approach is a bottom-up approach, being data (or database) centric. It is the best approach for extending the usefulness of existing, legacy data stored in relational databases. It has the best hope for addressing the issues of impedance mismatch and performance penalty that one encounters when accessing relational data from an **object**-oriented **programming** language. It, however, has the drawback of focusing only on data stored in relational databases or whatever in the future can be stored in extended relational databases. ORDBMSs address the impedance mismatch problem in relational DBMS by providing more and more support for the data modeling features of major **object**-oriented **programming** languages. However, the data model that is used by an application can be close but not identical to the data model used in the DBMS to store the application data. ORDBMSs make up for this problem by providing complex query capability and rich support to execute portions of the application within the database server. In addition, they have the best robustness, concurrency, and crash recovery characteristics among all three classes of systems. Applications that need extremely good query support, excellent security, integrity, concurrency, and robustness, and high transaction rates are best candidates for using an ORDBMS.

The OODBMS approach is a top-down approach, being application (or programming language) centric. It is the best approach for storing application objects, e.g., presentation or view objects. It has the best hope for providing seamless persistence, from a programming language point of view. OODBMSs avoid the impedance mismatch by providing extensive support for the data modeling features of one or more **object**-oriented **programming** languages. In an OODBMS, therefore, the data model that is used by an application is identical to the data model used in the DBMS to store the application data. However, OODBMSs do not provide as good a query facility as ORDBMS. Additionally, the transaction rates supported by the OODBMSs do not yet approach the high rates achieved by relational DBMSs on standard transaction processing benchmarks. Applications that need excellent navigational performance, that do not have complex query, and that are prepared to sacrifice some integrity and security for achieving good performance are best suited for using OODBMSs.

In the future, it is likely that we will see the continued presence of OODBMSs that address the needs of specialized markets, the continued prominence of ORDBMSs that address the needs of traditional commercial markets, and the growing importance and prevalence of the gateways-integrated with object query, object transaction and workflow, and object security (i.e., a full-function object middleware or multidatabase). It therefore becomes extremely important for an object-oriented application developer to choose the right type of system for storing objects. As evidenced by our discussion of the various issues, this task could be a fairly daunting one. We hope that our discussion provides valuable insight to developers to make it easier to choose the right persistent system for an object-oriented application.

Acknowledgments

Our thanks to the reviewers and the editor for giving us valuable comments.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Informix, Inc., Oracle, Inc., Sybase, Inc., Object Design, Inc., Persistence Software, Inc., UniSQL, Inc., Servio Corporation, Subtle Software, Inc., Objectivity, Inc., Ontos, Inc., 02 Technology Inc., Itasca, Inc., Matisse, Inc., Sun Microsystems, Inc., or Versant Object Technology.

[Footnote]

[Reference]
Cited references

[Reference]
1. C. Booch, Object-Oriented Analysis and Design with Applications, second edition, The Benjamin/Cummings Publishing Company, Redwood City, CA (1994).
2. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ (1991).
3. B. Stroustrup, The C+ + Programming Language, second edi

[Reference]
tion, Addison-Wesley Publishing Company, Reading, MA (1993).
4. W. Lalonde, Discovering Smalltalk, The Benjamin/Cummings
Publishing Company, Redwood City, CA (1994). 5. J. Gosling and H. McGilton, "The Java Language Environment: A White Paper," Sun Microsystems, http: //www.javasoft.com/whitePaper/javawhitepaper-l.html (1994).
6. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM 13, No. 6 (June 1970).
7. ISO-ANSI, Database Language SQL, ISO/IEC 9075:1992 (1991); American National Standards Institute,11 West 42nd Street, New York, NY 10036.
8. B. Reinwald, T. J. Lehman, H. Pirahesh, and V. Gottemukkala, "Storing and Using Objects in a Relational Database," IBM Systems Journal 35, No. 2, 172-191 (1996). 9. OMG, CORBAservices: Common Object Services Specifica

[Reference]

tion, Revised Edition, OMG Doc. No. 95-3-31 (March,1995). Object Management Group, Inc., 492 Old Connecticut Path.

Framingham, MA 01701.
10. ISO-ANSI, Working Draft Database Language SQL/ Foundation (SQL3), Part 2 X3H2-93-329, DBL RIO-004, Jim Melson, Editor (August,1994); American National Standards Institute, 11 West 42nd Street, New York, NY 10036. 11. The Object Database Standard: ODMG-93, Release 1.1, R. G. G. Cattell, Editor, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1994).
12. R. B. Murray, C+ + Strategies and Tactics, Addison-Wesley
Publishing Company, Reading, MA (1993). 13. F. Bancilhon, C. Delobel, and P. Kannellakis, Building an Object-Oriented System: The Story of 02, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1992).
14. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," Communications of the ACM 34, No. 10 (October 1991).
15. C. Lau, **Object**-Oriented **Programming** using SOM and DSOM, Van Nostrand Reinhold Publishing Company (1994).
16. M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures,"

[Reference]
Proceedings of the ACM SIGMOD Conference (1991). 17. J. Ullman, Principles of Database and Knowledge-Based Systems, Volumes 1 and 2, Computer Science Press, Rockville, MD (1989).
18. F. Bancilhon, W. Kim, and H. Korth, "A Model of CAD Transactions," Proceedings of the International Conference on Very Large Databases (1984).
19. R. H. Katz and T. Lehman, "Database Support for Versions and Alternatives of Large Design Files," IEEE Transactions on Software Engineering 10, No. 2 (1984).
General references

[Reference]
C. Alfred, "Maximizing Leverage from an Object Database," IBM Systems Journal 33, No. 2, 280-299 (1994). J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Database Systems," Proceedings of the ACM SIGMOD Conference (1987).
J. M. Cheng, N. M. Mattos, D. D. Chamberlin, and L. G. De Michiel, "Extending Relational Database Technology for New Applications," IBM Systems Journal 33, No. 2, 264-279 (1994). F. Leymann and W. Altenhuber, "Managing Business Processes

[Reference]
as an Information Resource," IBMSystems Joumal33, No. 2, 326348 (1994).
M. Schlatter, R. Furegati, F. Jeger, H. Schneider, and H. Streckeisen, "The Business Object Management System," IBM Systems Journal 33, No. 2, 239-263 (1994).
Accepted for publication August 23, 1996.

[Author Affiliation]
V. Srinivasan IBM Software Solutions Division, Santa Teresa Laboratory, 555 Bai/eyAvenue, San Jose, CaliJbnia 95141. Dr. Srinivasan is a member of the Database Technology Institute at the IBM Santa Teresa Laboratory. His interests include object databases, object-relational gateways, internet access to databases, and concurrency control and recovery algorithms for indexes. At IBM, he has contributed extensively to the design and implementation of ObjectStore/DB2 Gateway, ObjectLite, and DB2WWW. Dr. Srinivasan received his B. Tech. degree in computer science from the Indian Institute of Technology, Madras, and his M.S. and Ph.D degrees in computer science from the University of Wisconsin, Madison.

[Author Affiliation]
Daniel T. Chang IBM Software Solutions Division, Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: dtcha,,vnet. ibm.colit). Dr. Chang is a member of the Database Technology Institute at the IBM Santa Teresa Laboratory. His current interests focus on network data computing, particularly mobile data agents, Java mobile agents, and Java object data access. He was the IBM Software Solutions Division representative to the Object Management Group (OMG) Technical Committee. He coauthored the OMG Persistent Object Service, Relationships Service, Compound Life Cycle Addendum to the Life Cycle Service, and Query Service. He has contributed extensively to the design of the VisualAge C+ + Data Access Builder/Class Library and the DataBasic Data Access Class Library. Previously, Dr. Chang worked at the IBM Palo Alto Scientific Center, developed a concurrent **object**-oriented **programming** system named CORAL, and made major contributions to the IBM expert systems effort. He has filed five patents in object technology. Dr. Chang received a Ph.D. in computational chemistry from the University of Chicago. He has published several papers on object technology, expert systems, software engineering, and computer simulation.